APPENDIX B
NETWORK MANAGEMENT AND DIAGNOSTIC SERVICES

In addition to application message services, the LonTalk protocol provides network management services for installation and configuration of nodes, downloading of software, and diagnosis of the network. Message codes used by the LonTalk protocol are as follows:

Message Type	**Hexadecimal Message Codes**
Application Messages	0x00 – 0x3E
Foreign Messages	0x40 – 0x4E
Network Diagnostics Messages	0x50 – 0x5F
Network Management Messages	0x60 – 0x73
Router Configuration Messages	0x74 – 0x7E
Service Pin Message	0x7F
Network Variable Messages	0x80 – 0xFF

Response codes used by the LonTalk protocol are as follows:

Response Type	**Hexadecimal Message Codes**
Application Responses	0x00 – 0x3E
Response if node is off-line	0x3F
Foreign Responses	0x40 – 0x4E
Response if node is off-line	0x4F
Network Diagnostic Success	0x31 – 0x3F
Network Diagnostic Failure	0x11 – 0x1F
Network Management Success	0x21 – 0x3D
Network Management Failure	0x01 – 0x1D
Router Configuration Success	0x34 – 0x3E
Router Configuration Failure	0x14 – 0x1E
Network Variable Poll Responses	0x80 – 0xFF

Notice that response codes are not unique. They must be interpreted based on the original request.

Section 7.3 describes the use of application messages. A foreign message is a packet of another protocol embedded in the LonTalk protocol packet. The application program allocates space for foreign message and response codes.

Network Diagnostic Messages	Request Code	Success Response	Failed Response
Query Status	0x51	0x31	0x11
Proxy Command	0x52	0x32	0x12
Clear Status*	0x53	0x33	0x13
Query XCVR Status	0x54	0x34	0x14

* This is what the LonBuilder development workbench uses for a reset command.

Network Management Messages	Request Code	Success Response	Failed Response
Query ID	0x61	0x21	0x01
Respond to Query	0x62	0x22	0x02
Update Domain	0x63	0x23	0x03
Leave Domain	0x64	0x24	0x04
Update Key	0x65	0x25	0x05
Update Address	0x66	0x26	0x06
Query Address	0x67	0x27	0x07
Query Net Variable Config	0x68	0x28	0x08
Update Group Address Data	0x69	0x29	0x09
Query Domain	0x6A	0x2A	0x0A
Update Net Variable Config	0x6B	0x2B	0x0B
Set Node Mode	0x6C	0x2C	0x0C
Read Memory	0x6D	0x2D	0x0D
Write Memory	0x6E	0x2E	0x0E
Checksum Recalculate	0x6F	0x2F	0x0F
Wink	0x70	0x30*	0x10*
Memory Refresh	0x71	0x31	0x11
Query SNVT	0x72	0x32	0x12
Network Variable Fetch	0x73	0x33	0x13
Device Escape Code	0x7D	0x3D	0x1D

* Only for request/response messaging service.

Router Configuration Messages	0x74 – 0x7E
Router Config Success Responses	0x34 – 0x3E
Router Config Failed Responses	0x14 – 0x1E

Router messages are used by network management nodes to configure nodes that run the special router system image. They are not useful for application nodes, which will return the failed response.

Service Pin Message	0x7F	(Unsolicited message)
Network Variable Messages	0x80 – 0xFF	

Network variable messages can not be received by an application program using explicit messaging syntax. They are sent by updating network output variables in the application program, and are implicitly received by network input variables in the same connection. Polling of network variables is implemented with request/response service. For a discussion of network variable messages, see Section B.3.

Network management and network diagnostic messages may be delivered using Request/ Response service (except for *mode on-line*, *mode off-line* and *wink*, which are delivered to the application processor and do not have response data associated with them). Network management messages that do not have response data associated with them may also be delivered with the other classes of service, namely Acknowledged, Unacknowledged and Unacknowledged/Repeated. These messages are: *Respond to Query, Update Domain, Leave Domain, Update Key, Update Address, Update Group Address Data, Update Net Variable Config, Set Node Mode, Write Memory, Checksum Recalculate, Memory Refresh,* and *Clear Status.* If broadcast addressing is used with Acknowledged or Request/Response service, then only the first acknowledgment or response can be handled.

Application messages and network variable updates are delivered with the specified class of service. Note that most network management and network diagnostic messages may be authenticated (if the `nm_auth` bit is set in the Configuration Structure). Authentication never applies to *Query ID, Respond to Query, Query Status,* or *Proxy messages.*

For Neuron C programs, these message structures are defined in the include file NETMGMT.H.

In the following descriptions of the network management messages, the data structures named NM_*xxx*_request specify the data field of the outgoing message (following the code field). Similarly, the data structures named NM_*xxx*_response specify the data field of the corresponding response. Network management messages are sent like any other explicit message, either from a host microprocessor or from a Neuron Chip. The following example shows how a Neuron C program running on a Neuron Chip could use the *Read Memory* network management message (see Section B.1.5) to retrieve the 6-byte Neuron ID from another Neuron Chip at offset 0x0000 into the Read-Only Structure:

```
// Note: the following structure is in the header file.

#include <ADDRDEFS.H>
#include <ACCESS.H>
#include <NETMGMT.H>

    struct {
       enum {
       absolute             = 0,
       read_only_relative   = 1,
       config_relative      = 2,
       }  mode;
       unsigned long offset;
       unsigned count;
    } read_rq;                          //  a local copy of the request msg

    msg_tag read_mem_tag;               //  declare a destination address

    when (  reset  )  {
       msg_out.code = 0x6D;               //  Read-memory code
       read_rq.mode = read_only_relative;// Address mode
       read_rq.offset = 0x0000;           //  Address offset
       read_rq.count = 6;                 //  Byte count
       memcpy( msg_out.data,  &read_rq, sizeof( read_rq ) );
                                          //  Copy into msg_out data array
       msg_out.service = REQUEST;         // Expect a response
       msg_out.tag = read_mem_tag;        // Destination address
       msg_send( );                       // Send the message
    }

    unsigned neuron_id[ 6 ];           // Place to save the returned ID

    when( resp_arrives( read_mem_tag ) ) {

       memcpy( neuron_id, resp_in.data, 6 ),// copy the response data to a
                                             local variable
    }
```

The failed response is returned when the destination Neuron Chip can not process the message, for example when a table index is out of range, there is a memory failure when writing to EEPROM, or an attempt is made to violate read/write protection.

Sending Network Management or Diagnostic Messages

Most NM/ND commands are delivered using the request-response service type. A few are limited to acknowledged service type. Throughout the messages descriptions that follow, request-response is assumed unless otherwise mentioned.

When the node is configured for network management authentication, most NM/ND transactions must be authenticated in order to take effect. However, if a node is not in the configured state, the network management authentication bit is ignored. Before setting a node's state to configured, the network manager should ensure that the network management authentication bit is set to the desired state. Network management messages that do not require authentication to be executed are so noted.

The transmit transaction timer value of the client node must be extended to handle the lengthy delays involved with any command that alters EEPROM. When Neuron ID addressing is used, the node that receives the NM or ND message automatically extends the non-group receive transaction timer to about 8 seconds. This allows the non-group receive transaction timer to be tuned for normal application traffic without concerns for lengthy network management transactions.

Addressing

Neuron ID addressed messages are received regardless of the domain in which they are sent. Unconfigured nodes will also accept any subnet or domain-wide broadcast regardless of the domain. In either of these cases, acknowledgments and responses are returned on the domain in which the message was received with a source subnet/node pair of 0/0. Messages received in a domain in which the node is not a member (either because the node is unconfigured or is simply not in the domain) are termed as being received on a flexible domain. Some commands are not permitted under these circumstances and are noted below.

An advantage of using Neuron ID addressing for network management commands is that if a node were to accidentally go unconfigured (e.g., due to a checksum error resulting from a power cycle while changing configuration), the network manager would not lose its ability to communicate with the node.

However, a disadvantage of using Neuron ID addressing is that the extended receive transaction timeout could lead to subsequent false detection of duplicates. Therefore, it is recommended that, if possible, subnet/node addressing be used for network management activity. Neuron ID addressing should be used only for communicating with nodes that are not in the configured state.

Configuration Changes

The paradigm for making configuration changes is as follows:

1. Alter the node state or condition (optional).
2. Perform the change or changes. Most messages automatically update the configuration checksum. The exception is memory writes to the configuration data structure.
3. Update the configuration checksum if necessary.
4. Return to step 2 if more needs to be done.
5. Restore the node state or condition if changed in step 1.
6. Reset the node if communication parameter changes were made. Communication parameters are copied from on-chip EEPROM to RAM at node reset, so that the media access control processor can access them even during EEPROM writes.

Step 1 typically involves taking a node off-line (with step 5 putting a node on-line). This is to ensure that the application is not accessing configuration addressing information as it is changing. It may be acceptable to eliminate this step in some circumstances.

In addition to going off-line, the actual node state may be changed to unconfigured or hard-off-line (with step 5 restoring it to configured). This has the advantage that, were the node to reset during the update, it would come up with the application not running. A disadvantage of making a state change is that the node state is considered to be part of the application. If it is corrupted (e.g., due to a power cycle while the state is being changed), the node will come up applicationless. If the network manager does not have the application available to reload, this can be catastrophic for the node. It is also important to note that the node should not leave the applicationless state as a result of reconfiguration alone. If the node is initially applicationless, setting it to configured will probably cause it to crash.

Application Downloading

The paradigm for downloading an application is as follows:

1. Take the node off-line.
2. Alter the node state to applicationless.
3. If node went bypass off-line (see Section B.1.6) in step 1, reset the node.
4. Perform a sequence of write memory commands to load the application. Writes should be limited to 11 bytes and the checksum should not be computed after each write. The order and contents of the writes should be determined by the contents of a .NXE (not NEI) load file generated by the LonBuilder export utility, with two exceptions. If the first record has a length of one byte, then it should not be written until after all the other records have been written. This is because it contains the read/write protect bit which could prevent further downloading if it is set. Also, if a record has a data count of zero, then the node should be reset at this point.
5. If no record with a data count of zero was encountered in step 4 (example: 59030000FC), reset the node in order to cause the RAM to be partitioned according to the new application's requirements.
6. Compute the application checksum.
7. Enter the unconfigured state.

Before the node may be put into the configured state, the network manager should make sure that the domain table, address table, and network variable configuration table have known states. Note that after loading an application followed by loading of the configuration, a node comes up in the off-line condition. To get the application running, you should initiate an on-line request.

Bypass off-line is defined as the application checking for "offline" events directly and invoking "offline_confirm" to effect the off-line condition. Normally, going off-line is handled by the scheduler.

Note that node resets can take quite a while. The slower a node's input clock, the longer the reset. The more off-chip EEPROM or RAM, the longer the reset. Durations up to 18 seconds are possible in the worst case. See Section 4.6.4 for details of the reset process.

Node Resets or Power Cycles: If a node resets while a network management command is in progress, the reset will likely manifest itself as either a communication problem or a transaction failure. When EEPROM writes are involved, there is a significant probability that a location being modified at the time of the reset will become corrupted (most likely with the erase pattern of 0xFF).

Read/Write Protect Violations: If a node is read/write protected, attempts to write to the application code area are denied. The client can verify that a write memory attempt failed for this reason by reading the `read_write_protect` field of the `read_only_data` structure.

B.1 NETWORK MANAGEMENT MESSAGES

These messages are described in six main groups: node identification messages, domain table messages, address table messages, network variable-related messages, memory-related messages, and special-purpose messages.

B.1.1 Node Identification Messages

Query ID (Request/Response Only)

This message requests selected nodes to respond with a message containing their 48-bit unique ID and program ID. This message is normally broadcast during network installation to find specific nodes in the domain. It can be used to find unconfigured nodes or explicitly selected nodes, or nodes with specified memory contents at a specified address. This address may be specified absolutely, relative to the Read-Only Structure (see Section A.1), or relative to the Configuration Structure (see Section A.6). Only data within the Read-Only Structure, the SNVT Structures (see Section A.5) or the Configuration Structure may be matched. This can be used for example to find nodes with a particular channel ID or location string (in the Configuration Structure) or program ID string (in the Read-Only Structure). In a group request/response, the sender will accept only the first response.

Message declarations:

```
// setup request
    typedef struct {
        enum {
          unconfigured     = 0,
          selected         = 1,
          selected_uncnfg  = 2,  // note: selected and unconfigured
        } selector;              // with respect to an address
        enum {                   // Begin optional fields
          absolute         = 0,  // all with respect to address.
          read_only_relative= 1,
          config_relative  = 2,
        } mode;
        unsigned long offset;
        unsigned count;
        unsigned data[];
    } NM_query_id_request;

// setup response
    typedef struct {
        unsigned neuron_id[ NEURON_ID_LEN ]; // not optional
        unsigned id_string[ ID_STR_LEN ];
    } NM_query_id_response;
```

The first byte of the request message specifies which nodes are to respond, whether unconfigured nodes, selected nodes, or nodes that are both selected and unconfigured. A node may be selected with the Respond to Query message. Optional fields may be present in the message which specify an address mode, a byte count and an array of bytes which must match a part of the destination node's memory. For interoperability, the length of the matched region can be up to 11 bytes long. The matched region may be in the read-only structure, the configuration structure, the SNVT structure or the application RAM data variable area. The address mode specifies whether the address of the matched memory is an absolute address in the memory space of the Neuron Chip, an address relative to the read-only data structure (see Section A.1), or an address relative to the configuration data structure (see Section A.6). If the memory matching feature is not required, the optional fields must not be present in the message; the length of the data part of the message must be one. Setting the *count* field to zero does not disable the memory matching

feature. The response message contains the 6-byte Neuron ID and the 8-byte program ID. Authentication is never used with this message.

Respond to Query

This message explicitly selects or deselects a node to respond to a Query ID message. It can be used to determine network topology. Resetting the node clears the selection.

Message declaration:

```
typedef enum {
      disable    = 0,
       enable    = 1,
} NM_respond_to_query_request;
```

The request message consists of a single byte specifying whether the node should be selected or deselected. Authentication is never used with this message.

Service Pin Message (Unsolicited)

This is an unsolicited message sent by a node when the service pin is grounded. It contains the node's unique ID followed by the program ID.

Message declaration from `netmgmt.h`:

```
typedef struct {
      unsigned neuron_id[ NEURON_ID_LEN ];
      unsigned id_string[ ID_STR_LEN ];
} NM_service_pin_msg;
```

The message data contains the 6-byte unique Neuron ID assigned by the manufacturer of the Neuron Chip, followed by the 8-byte ID of the application program in the Neuron Chip. See Section A.1.1 for details on these values. The service pin message is sent as a domain-wide broadcast on the domain whose length is zero. The source subnet and node IDs are both zero. Routers retransmit service pin messages on the domains in which they are configured.

B.1.2 Domain Table Messages

Update Domain

This message overwrites a domain table entry with a new value, and recomputes the configuration checksum. This assigns a domain, subnet and node identifier to the node, as well as an authentication key for that domain. For security reasons, the authentication key should not be transmitted on an open network where it is possible for others to tap in the network and learn the authentication key value. The node does not enter the configured state until a Set Node Mode message is sent to change its state to configured. The Update Domain message is honored even if the node is read/write protected. For a description of the domain table, see Section A.2. If the Update Domain message is received by a node on the domain that is being updated, the response is returned in the new domain. A node that receives this message can take up to 330 ms to execute the function.

Message declaration from netmgmt.h:

```
typedef struct {
    unsigned domain_index;
    unsigned id[ DOMAIN_ID_LEN ];
    unsigned subnet;
    unsigned must_be_one    :1;// clone domain bit this bit must be set to 1
    unsigned node           :7;
    unsigned len;
    unsigned key[ AUTH_KEY_LEN ];
} NM_update_domain_request;
```

The request message consists of an index into the domain table (0 or 1), followed by an image of the domain table entry to be written, in the format of a domain_struct declared in \..\INCLUDE \ACCESS.H. The most significant bit of the byte containing the node ID must be set for normal operation. If this bit is clear, the node is treated as a "cloned" node. That is, the node will not be able to receive messages addressed to it using subnet/node address format. It will, however, be able to receive messages from another node that has the same subnet and node IDs in that domain.

Query Domain (Request/Response Only)

This message retrieves an entry in the domain table. This message is honored even if the node is read/write protected. For a description of the domain table, see Section A.2.

Message declarations:

```
typedef unsigned /* domain_index */ NM_query_domain_request;
```

```
typedef struct {
    unsigned id[ DOMAIN_ID_LEN ];
    unsigned subnet;
    unsigned                : 1;
    unsigned node           : 7;
    unsigned len;
    unsigned key[ AUTH_KEY_LEN ];
} NM_query_domain_response;
```

The request message consists of an index into the domain table (0 or 1). The response message contains an image of the domain table entry that was read, in the format of a domain_struct declared in \LB\INCLUDE\ACCESS.H.

Leave Domain

This message deletes a domain table entry, and recomputes the configuration checksum. After the message is processed, if the node does not belong to any domain, it becomes unconfigured and is reset. This message is honored even if the node is read/write protected. For a description of the domain table, see Section A.2. If the Leave Domain message is received by a node on the domain which is to be left, no response is returned. If the message causes the node to leave the last domain where it is configured, its state becomes unconfigured. A node that receives this message can take up to 330 ms to execute the function.

Message declaration from netmgmt.h:

```
typedef unsigned /* domain_index */ NM_leave_domain_request;
```

The request message consists of an index into the domain table (0 or 1).

Update Key

This message adds an increment to the current encryption key in a domain table entry to form a new key, and recomputes the configuration checksum. This allows rekeying of a node without having to transmit the new key on an open or public network. This message is honored even if the node is read/write protected. For a description of the domain table, see Section A.2. Senders of this message should be especially careful that the message is not received more than once, since its function is incremental. A node that receives this message can take up to 150 ms to execute this function.

Message declaration from `netmgmt.h`:

```
typedef struct {
    unsigned domain_index;
    unsigned key[ AUTH_KEY_LEN ];
} NM_update_key_request;
```

The request message consists of an index into the domain table (0 or 1), followed by six bytes of authentication key increment.

B.1.3 Address Table Messages

Update Address

This message overwrites an address table entry with a new value, and recomputes the configuration checksum. An address table entry allows the node to implicitly address another node, or join a group. This message is honored even if the node is read/write protected. For a description of the address table, see Section A.3. A node that receives this message can take up to 130 ms to execute the function.

Message declaration from `netmgmt.h`:

```
typedef struct {
    unsigned addr_index;
    unsigned type;                          //addr_type or (0x80 |group_size)
    unsigned domain         : 1;
    unsigned member_or_node : 7;
    unsigned rpt_timer      : 4;
    unsigned retry          : 4;
    unsigned rcv_timer      : 4;
    unsigned tx_timer       : 4;
    unsigned group_or_subnet;
} NM_update_addr_request;
```

The request message consists of an index into the address table (0 to 14), followed by an image of the address table entry to be written, in the format of an `address_struct` declared in `\..\INCLUDE\ACCESS.H`. The address type field is 0 for `unbound`, 1 for `subnet_node`, 3 for `broadcast`, and for group addressing it contains the group size with the most significant bit set.

Query Address (Request/Response Only)

This message retrieves an entry in the address table. This message is honored even if the node is read/write protected. For a description of the address table, see Section A.3.

Message declarations from netmgmt.h:

```
typedef unsigned /* addr_index */ NM_query_addr_request;

typedef struct {
    unsigned type;                      //addr_type or (0x80 |group_size)
    unsigned domain         : 1;
    unsigned member_or_node : 7;
    unsigned rpt_timer      : 4;
    unsigned retry          : 4;
    unsigned rcv_timer      : 4;
    unsigned tx_timer       : 4;
    unsigned group_or_subnet;
} NM_query_addr_response;
```

The request message consists of an index into the address table (0 to 14). The response message contains an image of the address table entry that was read, in the format of an address_struct declared in \LB\INCLUDE\ACCESS.H. The address type field is 0 for unbound, 1 for subnet_node, 3 for broadcast, and for group addressing it contains the group size with the most significant bit set.

Update Group Address Data

This message updates a group entry in the address table with a new group size and timer fields, and recomputes the configuration checksum. The message is sent to all members of the group, and updates the corresponding entry in the address table. This is used when nodes join or leave the group. This message is honored even if the node is read/write protected. For a description of the address table, see Section A.3. A node that receives this message can take up to 130 ms to execute the function.

Message declaration from netmgmt.h:

```
typedef struct {
    unsigned type           : 1;        //must be one
    unsigned size           : 7;
    unsigned domain         : 1;
    unsigned member         : 7;
    unsigned rpt_timer      : 4;
    unsigned retry          : 4;
    unsigned rcv_timer      : 4;
    unsigned tx_timer       : 4;
    unsigned group;
} NM_update_group_addr_request;
```

The request message consists of an image of the address table entry to be written, and must be delivered with group addressing. The group size and timer values are updated with new values, but the member number and domain index are left unchanged. The group number must not change.

B.1.4 Network Variable-Related Messages

Network variable update and poll messages are described in Section B.3.

For the network variable-related messages that have an optional 16-bit field following a one-byte network variable index *(Update Net Variable Config, Query Net Variable Config,* and *Network Variable Fetch),* the two optional bytes are only present if the one-byte index is 0xFF. This is valid only for host-based nodes. The two optional bytes are only necessary if the network variable index used is larger than 254.

Update Net Variable Config

This message overwrites a network variable configuration or address table entry with a new value, and recomputes the configuration checksum. This assigns a network variable selector to effect the binding of the network variable to network variables with the same selector on other nodes. This message is honored even if the node is read/write protected. For a description of the network variable configuration table, see Section A.4. For a node using a LONWORKS network interface with host selection enabled (ex: Microprocessor Interface Program with the network variable configuration table on the host), the *Update Net Variable Config* message is passed to the host microprocessor. In this case, the network variable index value of 255 is reserved to indicate that the following two bytes in the message form a 16-bit network variable index, allowing up to 16,383 network variables to be bound. Note, the MIP only supports up to and including 4096 network variables.

Values 0 – 0xFFF are valid network variable configuration table indices, allowing up to 4096 network variable table entries to be updated. For updating the network variable alias table, the actual index used is the index of the alias table plus the `nv_count`. Values (`nv_count`) – (`nv_count` + 0xFFF) are valid network variable alias table indices, allowing up to 4,096 network variable alias table entries to be updated. A Neuron Chip-hosted node that receives this message can take up to 110 ms to execute the function.

```
typedef struct {
    unsigned nv_index;
    unsigned nv_priority    : 1;
    unsigned nv_direction   : 1;
    unsigned nv_selector_hi : 6;
    unsigned nv_selector_lo : 8;
    unsigned nv_turnaround  : 1;
    unsigned nv_service     : 2;
    unsigned nv_auth        : 1;
    unsigned nv_addr_index  : 4;
} NM_update_nv_cnfg_request;
```

The request message consists of an index into the network variable table, followed by an image of the network variable configuration table entry to be written, in the format of an `nv_struct` declared in `\..\INCLUDE\ACCESS.H`.

Query Net Variable Config (Request/Response Only)

This message retrieves an entry in the network variable configuration table. This message is honored even if the node is read/write protected. For a description of the network variable configuration table and alias table, see Section A.4. For a node using a LONWORKS network interface with host selection enabled (ex: Microprocessor Interface Program with the network variable configuration table on the host), the *Query Net Variable Config* message is passed to the host microprocessor. In this case, the network variable index value of 255 is reserved to indicate that the following two bytes in the message form a 16-bit network variable index, allowing up to 16,383 network variable configuration table entries to be queried.

Values 0 – 0xFFF are valid network variable configuration table indices, allowing up to 4,096 network variable table entries to be updated. For updating the network variable alias table, the actual index used is the index of the alias table plus the `nv_count`. Values (`nv_count`) – (`nv_count` + 0xFFF) are valid network variable alias table indices, allowing up to 4,096 network variable alias table entries to be updated.

Message declaration from `netmgmt.h`:

```
typedef unsigned /* nv_index */         NM_query_nv_cnfg_request;

typedef struct {
    unsigned nv_priority    : 1;
    unsigned nv_direction   : 1;
    unsigned nv_selector_hi : 6;
    unsigned nv_selector_lo : 8;
    unsigned nv_turnaround  : 1;
    unsigned nv_service     : 2;
    unsigned nv_auth        : 1;
    unsigned nv_addr_index  : 4;
} NM_query_nv_cnfg_response;
```

The request message consists of an index into the network variable configuration table. The response message contains an image of the network variable configuration table entry that was read, in the format of an `nv_struct` declared in `\..\INCLUDE\ACCESS.H`.

Query SNVT (Request/Response Only)

This message retrieves self-identification and self-documentation data from the host processor memory of a node using a LONWORKS network interface (ex: Microprocessor Interface Program). The MIP is a special application used to attach the node to a host processor at Layer 4 of the protocol. In these cases, the address table is located in the Neuron Chip memory, but the network variable fixed table and SNVT information is located in the host's memory. The specified amount of data is retrieved from the specified offset into the SNVT Structure on the host. The number of bytes read in one message is limited by the network buffer sizes on both Neuron Chips. For interoperability, this should be limited to 16 bytes. For a Neuron Chip-hosted node, the *Query SNVT* message will return the failed response. In this case the *Read Memory* message should be used to retrieve the SNVT information using the `snvt` pointer in the Read-Only Structure (see Section A.1). For a description of the SNVT Structure, see Section A.5.

Message declarations from `netmgmt.h`:

```
typedef struct {
    unsigned long offset;
    unsigned count;
} NM_query_SNVT_request;

typedef unsigned NM_query_SNVT_response[];
```

The request message consists of two bytes specifying the offset into the addressed memory, and a byte specifying the number of bytes to be retrieved. The address is passed with the most significant byte first, whether or not this is the native address format of the host microprocessor. The response message contains the data in the memory that was read.

Network Variable Fetch (Request/Response Only)

This message retrieves the value of a network variable by its index into the network variable tables. This can be used to poll the value of a network variable, even if the node is off-line. For a description of the network variable tables, see Section A.4. The normal way to poll a network variable value is with an application message specifying the network variable's selector (see Section B.3). For a node using a LONWORKS network interface (ex: MIP), the *Network Variable Fetch* message is passed to the host microprocessor. In this case, the network variable index value of 255 is reserved to indicate that the following two bytes in the message form a 16-bit network variable index, allowing up to 16,383 network variables to be fetched. Only values 0 – 0xFFF are valid, allowing up to 4096 network variables to be fetched.

Message declarations from `netmgmt.h`:

```
typedef unsigned /* nv_index */ NM_NV_fetch_request;

typedef struct {
    unsigned nv_index;
    unsigned data[];
} NM_NV_fetch_response;
```

The request message consists of the network variable index, which is the index into either of the network variable tables. The response message contains the network variable index, followed by the network variable data itself.

B.1.5 Memory-Related Messages

Read Memory (Request/Response Only)

This message reads data from the node's memory. Addresses may be specified relative to the Read-Only Structure (see Section A.1), relative to the Configuration Structure (see Section A.6), relative to the query statistics structure (see below), or absolutely. To read the domain table, the address table, or the network variable configuration table, the *Query Domain/Address/NV Config* messages should be used. The number of bytes that can be read in one message is limited only by the network buffer sizes on both Neuron Chips. If the node is read/write protected, none of the node's memory may be read, except for the Read-Only Structure (see Section A.1), the SNVT Structures (see Section A.5), and the Configuration Structure (see Section A.6).

Message declaration from `netmgmt.h`:

```
typedef struct {
    enum {
        absolute                = 0,
        read_only_relative      = 1,
        config_relative         = 2,
        statistics_relative     = 3,
    } mode;
    unsigned long offset;
    unsigned count;
} NM_read_memory_request;

typedef unsigned NM_read_memory_response[];
```

The request message consists of a byte specifying the address mode, two bytes specifying the offset into the addressed memory (most significant byte of the address first), and a byte specifying the number of bytes to be read. The response message contains the data in the memory that was read.

The following structure definition should be used for a read memory command with statistics-relative addressing mode:

```
typedef struct {
    unsigned long     transmission_errors;
    unsigned long     transmit_tx_failures;
    unsigned long     receive_tx_full;
    unsigned long     lost_messages;
    unsigned long     missed_messages;
    unsigned long     layer2_received;
    unsigned long     layer3_received;
```

```
    unsigned long     layer3_transmitted;
    unsigned long     transmit_tx_retries;
    unsigned long     backlog_overflows;
    unsigned long     late_acknowledgments;
    unsigned long     collisions;
    unsigned int      eeprom_lock;
} stats_struct;
```

The fields for the above structure are defined as follows:

```
    unsigned long     transmission_errors;
```

The number of CRC errors detected during packet reception. These may be due to collisions or noise on the transceiver input.

```
    unsigned long     transmit_tx_failures;
```

The number of times that the node failed to receive expected acknowledgments or responses after retrying the configured number of times. These may be due to destination nodes being inaccessible on the network, transmission failures because of noise on the channel, or if any destination node has insufficient buffers or receive transaction records. When using request/response service or network variable polling, a transaction timeout can also occur if the destination node application program does not return to the scheduler frequently enough because responses are synchronized with the application tasks.

```
    unsigned long     receive_tx_full;
```

The number of times that an incoming packet was discarded because there was no room in the transaction database. These may be due to excessively long receive timers (Section A.3.11), or inadequate size of the transaction database.

```
    unsigned long     lost_messages;
```

The number of times that an incoming packet was discarded because there was no application buffer available. These may be due to an application program being too slow to process incoming packets, to insufficient application buffers, or to excess traffic on the channel. If the incoming message is too large for the application buffer, an error is logged, but the lost message count is not incremented.

```
    unsigned long     missed_messages;
```

The number of times that an incoming packet was discarded because there was no network buffer available. These may be due to excess traffic on the channel, to insufficient network buffers, or to the network buffers not being large enough to accept all packets on the channel, whether or not addressed to this node.

```
    unsigned long     layer2_received;
```

The number of layer-2 messages received by this node. Layer-2 messages are those that have correct CRC and can be addressed to any node.

```
    unsigned long     layer3_received;
```

The number of layer-3 messages received by this node. Layer-3 messages are those layer-2 messages that are addressed to this node.

```
    unsigned long     layer3_transmitted;
```

The number of messages transmitted from layer 3 of the Neuron Chip. These can include network variable updates, explicit messages, acknowledgments, retries, reminders, service pin messages, and any other type of message.

```
unsigned long     transmit_tx_retries;
```

The number of retries sent by this node. This does not include retries used for messages sent with the repeated service.

```
unsigned long     backlog_overflows;
```

The number of times the backlog reached its maximum value of 63.

```
unsigned long     late_acknowledgments;
```

The number of acknowledgments or responses that arrived at this node after the transmit transaction had expired.

```
unsigned long     collisions;
```

This field specifies the number of occurrences of either collision detection or collision resolution, if those features are enabled.

```
unsigned int      eeprom_lock;
```

The state of the EEPROM lock for the node. If this field is a one, then the checksummed EEPROM on the node is protected against memory write.

Write Memory

This message writes data to the node's memory. Addresses may be specified relative to the Read-Only Structure (see Section A.1), relative to the Configuration Structure (see Section A.6), or absolutely. This message may be used to download an application program to read/write memory on the node. The number of bytes that can be written in one message is limited by the network buffer sizes on both Neuron Chips. To write the domain table, the address table, or the network variable configuration table, the *Update Domain/Address/NV Config* messages should be used. The number of bytes that can be written in one message is limited by the network buffer sizes on both Neuron Chips. For interoperability, this should be limited to 11 bytes. If writing to EEPROM, the application checksum and the configuration checksum may be recalculated. In this case, the number of bytes written in one message should be limited to 38 to avoid watchdog timeouts at maximum input clock rate. The node may also be reset. If the node is read/write protected, none of the node's memory may be written except for the Configuration Structure.

Message declaration:

```
typedef struct {
    enum {
        absolute                = 0,
        read_only_relative      = 1,
        config_relative         = 2,
        statistics_relative     = 3,
     } mode;
    unsigned long offset;
     unsigned count;
    enum {
        no_action               = 0,
        both_cs_recalc          = 1,
        cnfg_cs_recalc          = 4,
        only_reset              = 8,
        both_cs_recalc_reset    = 9,
        cnfg_cs_recalc_reset    = 0xC,
     } form;
    unsigned data[];
} NM_write_memory_request;
```

The request message consists of a byte specifying the address mode, two bytes specifying the offset into the addressed memory (most significant byte of the address first), a byte specifying the number of bytes to be written, and a byte specifying the action to be taken at the completion of the write operation, followed by the data bytes to be written.

The following structure definition should be used for a read memory command with statistics-relative addressing mode:

```
typedef struct {
    unsigned long     transmission_errors;
    unsigned long     transmit_tx_failures;
    unsigned long     receive_tx_full;
    unsigned long     lost_messages;
    unsigned long     missed_messages;
    unsigned long     layer2_received;
    unsigned long     layer3_received;
    unsigned long     layer3_transmitted;
    unsigned long     transmit_tx_retries;
    unsigned long     backlog_overflows;
    unsigned long     late_acknowledgments;
    unsigned long     collisions;
    unsigned long     eeprom_lock
} stats_struct;
```

The fields for the above structure are defined as follows:

```
    unsigned long     transmission_errors;
```

The number of CRC errors detected during packet reception. These may be due to collisions or noise on the transceiver input.

```
    unsigned long     transmit_tx_failures;
```

The number of times that the node failed to receive expected acknowledgments or responses after retrying the configured number of times. These may be due to destination nodes being inaccessible on the network, transmission failures because of noise on the channel, or if any destination node has insufficient buffers or receive transaction records. When using Request/Response service or network variable polling, a transaction timeout can also occur if the destination node application program does not return to the scheduler frequently enough because responses are synchronized with the application tasks.

```
    unsigned long     receive_tx_full;
```

The number of times that an incoming packet was discarded because there was no room in the transaction database. These may be due to excessively long receive timers (see Section A.3.11), or inadequate size of the transaction database.

```
    unsigned long     lost_messages;
```

The number of times that an incoming packet was discarded because there was no application buffer available. These may be due to an application program being too slow to process incoming packets, to insufficient application buffers, or to excess traffic on the channel. If the incoming message is too large for the application buffer, an error is logged, but the lost message count is not incremented.

```
unsigned long     missed_messages;
```

The number of times that an incoming packet was discarded because there was no network buffer available. These may be due to excess traffic on the channel, to insufficient network buffers, or to the network buffers not being large enough to accept all packets on the channel, whether or not addressed to this node.

```
unsigned long     layer2_received;
```

The number of layer-2 messages received by this node. Layer-2 messages are those that have correct CRC and can be addressed to any node.

```
unsigned long     layer3_received;
```

The number of layer-3 messages received by this node. Layer-3 messages are those layer-2 messages that are addressed to this node.

```
unsigned long     layer3_transmitted;
```

The number of messages transmitted from layer 3 of the Neuron Chip. These can include network variable updates, explicit messages, acknowledgments, retries, reminders, service pin messages, and any other type of message.

```
unsigned long     transmit_tx_retries;
```

The number of retries sent by this node. This does not include retries used for messages sent with the repeated service.

```
unsigned long     backlog_overflows;
```

The number of times the backlog reached its maximum value of 63.

```
unsigned long     late_acknowledgments;
```

The number of acknowledgments or responses that arrived at this node after the transmit transaction had expired.

```
unsigned long     collisions;
```

This field specifies the number of occurrences of either collision detection or collision resolution, if those features are enabled.

```
unsigned int      eeprom_lock;
```

The state of the EEPROM lock for the node. If this field is a one, then the checksummed EEPROM on the node is protected against memory write.

Checksum Recalculate

This message recomputes either the network image checksum, or both the network and application image checksums in EEPROM. The application image and the network image are independently checked whenever the node is reset. They are also checked by a continually running background task. If the configuration checksum is invalid, the node enters the unconfigured state. If the application checksum is invalid, the node enters the application-less state. The LonBuilder compiler will generate the configuration and application checksums. When a program is loaded over the network, these two checksums are also loaded. After reset and before program execution the checksums are verified. Refer to Section 4.6.4, Reset Processes and Timing, for more information.

The network variable messages to update the domain, address, or network variable tables automatically update the configuration checksum. The *Checksum Recalculate* message is intended for use after a series

of *Write Memory* messages, for example, when downloading an application program. The checksum recalculate is used after a LonBuilder memory write command.

Message declaration:

```
typedef enum {
    both_cs = 1,            // Application image and network image checksums
    cnfg_cs = 4,            // Network image checksum only
} NM_checksum_recalc_request;
```

The request message consists of a single byte specifying which checksums should be recalculated.

Memory Refresh

This message rewrites EEPROM memory at the specified offset. Either on-chip or off-chip EEPROM may be refreshed. The number of bytes refreshed in one message should be limited to 38 if the node is off-line, 8 bytes if the node is on-line, to avoid watchdog time outs at maximum input clock rates. This message may be used periodically to extend the 10-year data retention of most EEPROM devices. Note that most EEPROM devices are specified as supporting 10,000 write cycles, therefore this message should not be used very frequently. The 48-bit Neuron ID can not be refreshed. The *Memory Refresh* message returns the failed response if the address specified is outside of the on-chip or off-chip EEPROM regions.

Message declaration from `netmgmt.h`:

```
typedef struct {
    unsigned long offset;
    unsigned count;
    enum {
        ON_chip  = 0,
        OFF_chip = 1,
     } which;
} NM_memory_refresh_request;
```

The request message consists of two bytes specifying the offset into the addressed memory (most significant byte of the address first), a byte specifying the number of bytes to be refreshed, and a byte indicating whether on-chip or off-chip EEPROM is to be refreshed.

B.1.6 Special-Purpose Messages

Set Node Mode (Service class varies)

A Neuron Chip's condition consists of two parts: state and mode. Generally, the node state is preserved across resets, and node mode is not. This message puts the application in an off-line mode, changes the state of the node, or resets the node.

A Neuron Chip can be in one of six states. These states are maintained in EEPROM and are as follows:

Node State	State Code	Service LED
Applicationless and Unconfigured	3	On
Unconfigured (but with an Application)	2	Flashing
Configured, Hard Off-Line	6	Off
Configured	4	Off

Applicationless and Unconfigured (3): No application is loaded yet, the application is in the process of being loaded, or the application is deemed corrupted due to application checksum error or signature inconsistency. The application does not run in this state. The service LED is steadily on in this state.

Unconfigured (2): The application is loaded but the configuration either is not loaded, is being reloaded, or is deemed corrupted due to configuration checksum error. A program can make itself unconfigured by calling the `go_unconfigured ()` function. The service LED flashes at a one second rate in this state.

Configured, Hard Off-Line (6): The application is loaded but not running. The configuration is considered valid in this state; the network management authentication bit is honored. The service LED is off in this state.

Configured (4): Normal Node State. The application is running and the configuration is considered valid. This is the only state in which messages addressed to the application are received. In all other states, they are discarded. The service LED is off in this state.

The configured state has an additional modifier, which is the on-line/off-line mode. This mode is **not** maintained in EEPROM. The states and on-line/off-line condition are controlled via different mechanisms. However, they are reported together in the *Query Status* network management message. The configured/off-line condition is also known as soft off-line.

A node that is in the soft off-line state will go on-line when it is reset. The hard off-line state is preserved across a reset. When the application is of either type off-line, the scheduler is disabled. When soft off-line, polling a network variable will return null data, but incoming network variable updates will be processed normally, except that the `nv_update_occurs` events will be lost. In all states other than configured, no response is returned to network variable polls and incoming network variable updates are discarded.

If a node is in a non-configured state, is reset, and is then issued a command to go configured, it will come up in a soft off-line condition.

If a set node mode message changes the mode to off-line or on-line, the appropriate Neuron C task (if any) will be executed. *Mode on-line* and *mode off-line* messages must not be delivered with request/response service. There will be no response to a *Reset* message, since the node is reset immediately. Changing the state of a node recomputes the configuration checksum, which takes some time, so verification of state changes should always be made with the *Query Status* message.

Message declaration from `netmgmt.h`:

```
typedef struct {
    enum {
        appl_offline            = 0,    // soft offline state
        appl_online             = 1,
        appl_reset              = 2,
        change_state            = 3,
    } mode;
    enum {
        appl_uncnfg             = 2,
        no_appl_uncnfg          = 3,
        cnfg_online             = 4,
        cnfg_offline            = 6,    // hard offline state
        } node_state;                   // Optional field if mode = 3
} NM_set_node_mode_request;
```

The request message consists of a byte specifying whether this is a request to put the node in the soft off-line state or on-line mode, reset it, or change the state of the Neuron Chip. In the last case, there is a second byte specifying which state the node should enter.

Wink (Any Service Class Except Request/Response)

This message has two formats. If the message is sent with no data, it causes the receiving node to execute the *wink* clause in the application program (if any). This may be used to identify nodes visually or audibly if it is more convenient than grounding the service pin. Wink messages may not be delivered with request/response service.

An example on a node receiving a wink message to turn on (off) the service LED is:

```
boolean service status;             // state of service pin
when (wink)
{
    service_status = 1 – service_status;         // keeping track of
                                                 // current STATE of
                                                 // service pin LED.
    activate_service_led = service_status;       // set service
                                                 // pin output.
}
```

Do not confuse the state of the service pin LED when the wink command is sent. The service pin LED is used, in this case, to signify reception of a wink command. Once the node is identified, the Neuron 48-bit ID can be extracted. Toggle the service pin again to put the service LED back into its original state.

The second format of this message is used only with host-based nodes, that is; nodes implemented with a LONWORKS network interface on a host processor. This format is needed when installing application nodes with multiple network interfaces attached. When installing a host-based node, depressing a service pin results in a single service pin message being generated. A network management toll can send this command to interrogate the node about any additional network interfaces that might be attached.

The first byte of the host-node format specifies a sub-command, which may be either a wink message or a request to send identifying information from the host. The `wink` subcommand may not be delivered with request/response service. The `send_id_info` subcommand should be delivered with request/response service, and the following byte in the message specifies a network interface number to support hosts that may have multiple network interfaces.

For host-based nodes, all forms of this message are passed to the host application, where they should be handled appropriately, either by executing a wink function or responding with the Neuron ID and program ID for the specified network interface. The first byte of the response should be zero if the specified network interface is functional, non-zero otherwise.

Message declaration from `netmgmt.h`:

```
typedef struct {                // used for host-based nodes only
    enum {
        WINK          = 0,
        SEND_ID_INFO  = 1,
    } subcommand;
    unsigned network_interface; // present if subcommand = SEND_ID_INFO
} NM_wink_request;

typedef struct {                // response to SEND_ID_INFO from a host
                                   node
    boolean interface_down;
    unsigned neuron_id[ NEURON_ID_LEN ];
    unsigned id_string[ ID_STR_LEN ];
} NM_wink_response;
```

B.2 NETWORK DIAGNOSTIC MESSAGES

Query Status (Request/Response Only)

This message retrieves the network error statistics accumulators, the cause of the last reset, the state of the node, and the last run-time error logged. This message is used after a node has been reset to verify that the reset has occurred, since resets are not acknowledged.

Message declaration:

```
typedef struct {
                unsigned long xmit_errors;          // offset 0x00
                unsigned long transaction_timeouts; // offset 0x02
                unsigned long rcv_transaction_full; // offset 0x04
                unsigned long lost_msgs;            // offset 0x06
                unsigned long missed_msgs;          // offset 0x08
                unsigned reset_cause;               // offset 0x0A
                unsigned node_state;                // offset 0x0B
                unsigned version_number;            // offset 0x0C
  enum {
      appl_uncnfg               = 2,
      no_appl_uncnfg            = 3,
      cnfg_online               = 4,
      cnfg_offline              = 6,   // hard offline state
      soft_offline              = 0xC,
      cnfg_bypass               = 0x8C
  } node_state;                         // offset 0x0C
   unsigned version_number;
  enum {
      no_error                                  = 0,
      bad_event                                 = 129,
      nv_length_mismatch                        = 130,
      nv_msg_too_short                          = 131,
      eeprom_write_fail                         = 132,
      bad_address_type                          = 133,
      preemption_mode_timeout                   = 134,
      already_preempted                         = 135,
      sync_nv_update_lost                       = 136,
      invalid_resp_alloc                        = 137,
      invalid_domain                            = 138,
      read_past_end_of_msg                      = 139,
      write_past_end_of_msg                     = 140,
      invalid_addr_table_index                  = 141,
      incomplete_msg                            = 142,
      nv_update_on_output_nv                    = 143,
      no_msg_avail                              = 144,
      illegal_send                              = 145,
      unknown_PDU                               = 146,
      invalid_nv_index                          = 147,
      divide_by_zero                            = 148,
      invalid_appl_error                        = 149,
      memory_alloc_failure                      = 150,
      write_past_end_of_net_buffer              = 151,
      appl_cs_error                             = 152,
```

```
        cnfg_cs_error                           = 153,
        invalid_xcvr_reg_addr                   = 154,
        xcvr_reg_timeout                        = 155,
        write_past_end_of_appl_buffer           = 156,
        io_ready                                = 157,
        self_test_failed                        = 158,
        subnet_router                           = 159,
        Authentication_mismatch                 = 160,
        self_inst_semaphore_set                 = 161,
        read_write_semaphore_set                = 162,
        appl_signature_bad                      = 163,
        router_firmware_version_mismatch        = 164,
    } error_log;                        // offset 0x0D
    unsigned model_number;              // offset 0x0E
} ND_query_status_response;
```

The request message contains no data. The response message begins with five 16-bit error statistics accumulators as follows:

Transmission errors — The number of CRC errors detected during packet reception. These may be due to collisions or noise on the transceiver input.

Transaction timeouts — The number of times that the node failed to receive expected acknowledgments or responses after retrying the configured number of times. These may be due to destination nodes being inaccessible on the network, transmission failures because of noise on the channel, or if any destination node has insufficient buffers or receive transaction records. When using Request/Response service or network variable polling, a transaction timeout can also occur if the destination node application program does not return to the scheduler frequently enough, because responses are synchronized with the application tasks.

Receive transaction full errors — The number of times that an incoming packet was discarded because there was no room in the transaction database. This may be due to excessively long receive timers (see Section A.3.11), or inadequate size of the transaction database.

Lost messages — The number of times that an incoming packet was discarded because there was no application buffer available. This may be due to an application program being too slow to process incoming packets, insufficient application buffers, or excess traffic on the channel. If the incoming message is too large for the application buffer, an error is logged, but the lost message count is not incremented.

Missed messages — The number of times that an incoming packet was discarded because there was no network buffer available. This may be due to excess traffic on the channel, insufficient network buffers, or the network buffers not being large enough to accept all packets on the channel, whether or not addressed to this node.

The response message also contains a byte with the cause of the last reset as follows (X = don't care):

Power-up reset	`0bXXXXXXX1`
External reset	`0bXXXXXX10`
Watchdog reset	`0bXXXX1100`
Software reset	`0bXXX10100`
Cleared	`0b00000000`

This is followed by a byte containing the current state and mode of the node. See Section B.1.6 for details on the node STATES. The byte may be:

Node Condition (State and Mode)	State/Mode Code	Service LED
Applicationless and Unconfigured	0x03	On
Unconfigured (but with an Application)	0x02	Flashing
Configured, Hard Off-Line	0x06	Off
Configured, Soft Off-Line	0x0C	Off
Configured, Bypass Off-Line	0x8C	Off
Configured, On-Line	0x04	Off

The state/mode byte has the following bit assignments:

State/Mode Byte

B	x	x	x	M	S	S	S

S = state
M = mode
B = bypass
x = not used

The mode and bypass bits are meaningful only when the state bits reflect the configured state.

The next byte in the response message indicates the version number of the firmware executing on the target node. For the firmware distributed with LonBuilder 2.0, this is 2. The version number is followed by a byte indicating the reason for the last error detected by the firmware on the target node. Zero means that no error has been detected since the last reset. For a description of the firmware errors, see the *Neuron C Programmer's Reference,* Chapter 11. LonBuilder and NodeBuilder support all versions of firmware from version 3 on, and also custom system images for Neuron 3150 Chips. Custom system images consist of a standard Neuron Chip firmware image combined with a custom system extension; they have version numbers in the range 128 – 254.

For custom system images (version numbers > 63), the version returned by the *Query Status* command can not be used to determine the original firmware version number since it reflects the version number assigned by the creator of the custom image. In order to obtain the original firmware version number on which the custom image was based, a single memory byte read of location 0x0000 must be performed over the network. This will return the original firmware version number. If the original firmware version reported is greater than 63, then the original firmware version can not be determined. Firmware version numbers 63 – 127 are reserved for use by Echelon. Any user developed custom system images should be assigned a firmware version number of 128 or greater.

The version number is followed by a byte indicating the reason for the last error detected by the firmware on the target node. This error number is stored in EEPROM. Zero means that no error has been detected since the last reset. Errors 134, 135, 150, and 151 cause the node to reset itself. For a description of the firmware errors, see *Neuron C Reference Guide,* Chapter 11.

The last byte in the query status response is the Neuron Chip model number: zero for a Neuron 3150 Chip, and eight for a Neuron 3120 Chip.

Clear Status

This message clears the network error statistics accumulators and the error log. The request message contains no data.

Proxy Command (Request/Response Only)

This message requests the node to deliver a *Query ID, Query Status,* or *Query Transceiver Status* message to another node. This can be used when physical channel limitations prevent a message from the network

management node from being received directly by the target tool (for example; when the target node is out of reach of the network management node). The response is also relayed back to the original sender.

Message declaration:

```
typedef struct {
    enum {
        query_unconfigured      = 0,
        status_request          = 1,
        xcvr_status             = 2,
    } sub_command;
    unsigned type;                      // addr_type or (0x80 |group_size)
    unsigned                    : 1;
    unsigned member_or_node     : 7;
    unsigned rpt_timer          : 4;
    unsigned retry              : 4;
    unsigned tx_timer           : 4;
    unsigned group_or_subnet;
    unsigned neuron_id[ 6 ];            // for type = 2
} ND_proxy_request;
```

The request message contains a byte specifying which message is to be delivered by proxy, followed by a destination address in the format of a `msg_out_addr` declared in `\..\INCLUDE\MSG_ADDR.H`. See Section A.3 for a description of destination address formats. The proxy message is delivered on the domain on which it was received. The address type field is 1 for `subnet_node`, 2 for `neuron_id`, 3 for `broadcast`, and for group addressing it contains the group size with the most significant bit set. The response message is the response appropriate to the specified status request. The node requesting the proxy service must take into account the response time through the agent when setting the transaction timer.

Query Transceiver Status (Request/Response Only)

This message retrieves transceiver status registers. This is a group of seven registers implemented in special-purpose mode transceivers. Even if the transceiver implements fewer than seven registers, seven values are returned in the response (see Section 4).

Message declaration from `netmgmt.h`:

```
typedef struct {
    unsigned xcvr_params[ NUM_COMM_PARAMS ];
} ND_query_xcvr_response;
```

The request message contains no data. The response message contains seven bytes with the transceiver status register values. If the transceiver of the node receiving this message does not support status registers, the response will have the fail bit set.

B.3 NETWORK VARIABLE MESSAGES

Network variable messages are of two types — network variable updates and network variable polls. All network variable messages are implemented with message codes in the range 0x80 – 0xFF, i.e., the most significant bit of the code is set.

A network variable update message is sent whenever an output network variable is updated by the application program, and the variable has been declared without the *polled* qualifier. These messages may be sent with Acknowledged, Unacknowledged, or Unacknowledged/Repeated service class. Updating an output network variable that has been declared with the *polled* qualifier does not cause a network variable update to be sent. A network variable update message contains the selector of the network variable that was updated, along with the data value. When a network variable update message is addressed to a node that has an input network variable whose selector matches the network variable selector in the message, then a network variable update event occurs on the destination node, and the value of the input network variable is modified with the data in the message. For a node using a LONWORKS network interface, the comparison of selectors may be done in the Neuron Chip if host selection is disabled, or in the host processor if host selection is enabled.

A network variable poll message is sent when the application program calls the *poll( )* system function specifying one or all of its input network variables. This message is sent with request/response service, and contains the selector of the polled network variable. When a network variable poll message is addressed to a node which has an output network variable whose selector matches the network variable selector in the message, then that node responds with a message containing the data in the network variable. This response is treated the same as a network variable update message; a network variable update event occurs on the requesting node, and the value of the input network variable is modified with the data in the message.

Normally, network variable updates and polls are delivered using implicit addressing, namely using an entry in the address table of the source node as the destination address. However, for special applications, it is possible to explicitly address network variable updates and polls by sending explicit messages that have the same structure as network variable messages.

Network Variable Update (Acknowledged, Unacknowledged, or Unacknowledged/Repeated)

Normally, network variables are updated across the network using implicit addressing. When the application program on the source node updates the value of a bound output (non-polled) network variable, the Neuron Chip firmware automatically builds an outgoing network variable update message using the information from the network variable configuration table and the address table. The information in these two tables is created as a result of the binding process. In certain applications, it is desirable to explicitly address a network variable update, rather than have the address implicitly taken from the network variable configuration and address tables of the source node. For example, if a single node wishes to send network variable updates to more than 15 different destination addresses (single nodes or groups), then it can use explicit addressing to overcome the limit of 15 address table entries on a node.

In this case, the source node can create an explicit message that is functionally identical to a network variable update message (see Figure B–1). The code field of this message contains the most significant six bits of the network variable selector. The most significant bit of the code field is set, indicating a network variable message, and the second most significant bit is clear, indicating that the update is addressed to an input network variable. The first data byte of the message contains the least significant eight bits of the network variable selector, and this is followed by the data in the network variable itself. The length of the data in the message must match the length of the destination network variable.

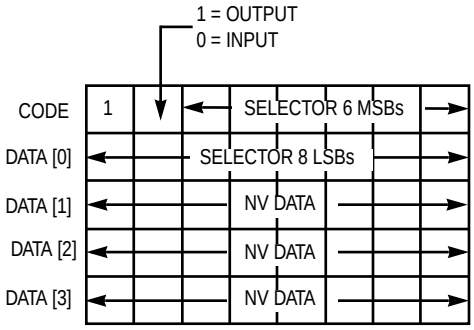

Figure B–1. Network Variable Message Structure

Example of updating a two-byte network variable whose selector is `0x1234` with the data value `0x5678`:

```
msg_out.code = 0x80 | 0x12;    // code field = 0x80 | nv_selector_hi
msg_out.data[ 0 ] = 0x34;      // data field = nv_selector_lo
msg_out.data[ 1 ] = 0x56;      // high byte of network variable value
msg_out.data[ 2 ] = 0x78;      // low byte of network variable value
```

Note that a network variable update message is processed by the network processor on the destination node. In a normal application program, the network variable update can not be received in the application processor using explicit messaging syntax. If an application needs to extract the source address from an incoming network variable update message, the Neuron C `nv_in_addr` built-in variable can be used (see Section 7.2). Nodes using a LONWORKS network interface can optionally receive the network variable update on the host processor.

When a network variable update is addressed using group (multicast) addressing with acknowledged service, all members of the group acknowledge the update message. Those members of the group that have input network variables with a matching selector will update those variables as a result of receiving the message, and generate an `nv_update_occurs` application event. Those members of the group that have an output network variable with a matching selector, or no network variable with a matching selector, will not generate any application event, even though they acknowledge the update message. If all the acknowledgments are successfully received by the sending node, an `nv_update_succeeds` event is generated. If one or more acknowledgments are not received after the configured number of retries, an `nv_update_fails` event is generated.

Network Variable Poll (Request/Response Only)

Normally, network variables are polled across the network using implicit addressing. When the application program on the source node issues a `poll( )` request to a bound input network variable, the Neuron Chip firmware automatically builds an outgoing network variable request message using the information from the network variable configuration table and the address table. The information in these two tables is created as a result of the binding process. In certain applications, it is desirable to explicitly address a network variable poll, rather than have the address implicitly taken from the network variable configuration and address tables of the source node. For example, if a single node wishes to poll network variables on more than 15 different destination nodes (single nodes or groups), then it can use explicit addressing to overcome the limit of 15 address table entries on a node. It can use a single input network variable to receive an unlimited number of responses to polls of any given data type.

In this case, the source node can create an explicit message that is functionally identical to a network variable poll message. The response to the poll will be processed by the network processor, and can not be received in the application program using explicit messaging syntax. If the node sending the poll message has an input network variable with the same selector and the same size as the polled network

variable, then this network variable will be updated by the response to the poll. This will generate nv_update_occurs events. A more convenient method of reading the value of a network variable with an explicitly addressed message is with the *Network Variable Fetch* network management message described in Section B.1.4. In this case, the response is processed by the application processor with one of the request/response functions and not the network processor. Alternatively, a node using a LONWORKS network interface can handle the response to the poll explicitly on the host processor if the MIP is configured with network variable processing off.

The code field of the *Network Variable Poll* request message contains the most significant six bits of the network variable selector. The most significant bit of the code is set, indicating a network variable message, and the second most significant bit is set indicating that the poll is addressed to an output network variable. The first data byte of the request message contains the least significant eight bits of the network variable selector. The response message contains the same code, except that the second most significant bit is clear, indicating that the response is addressed to an input network variable. The first data byte of the response contains the least significant eight bits of the network variable selector, and this is followed by the data in the network variable itself. If the poll is received by a node that has no matching network variable, or the node is offline, then the response contains the selector, but no data is present.

When a network variable poll is addressed using group (multicast) addressing with acknowledged service, all members of the group acknowledge the poll request message. Those members of the group that have output network variables with a matching selector will respond with a message containing the value of the variable. These responses will generate nv_update_occurs events on the polling node. Those members of the group that have an input network variable with a matching selector, or no network variable with a matching selector, or are off-line, will generate a response containing no data. The generation of these responses requires the participation of the application processor in the polled node, and occurs at the end of the currently executing critical section. This should be taken into account when designing the application code for a node whose network variables may be polled, and when configuring the transaction timer for the poll message. If all the responses are successfully received by the polling node, an nv_update_succeeds event is generated. If one or more responses is not received after the configured number of retries, an nv_update_fails event is generated.